Implementation of a Dominoes Game

- V2 -

Author:
Patricia Reinoso

Adviser:
J. Paul Gibson

Télécom SudParis
MSc. Computer Science for Communication Networks
Object Oriented Computing and Distributed Systems

May 10, 2017

Contents

Chapter 1

Introduction

In order to master the object-oriented paradigm, understanding design patterns, applying some advanced programming techniques, and learning features of the Java language, a simulation of domino game was developed.

The simulation consisted on a standard version of the game between a computer and an user player. To accomplish this task, a graphical interface, some documentation were provided during a previous module of the course.

The new assignment consisted on improving or making changes on the first version, incorporating some advanced techniques learnt on the lectures. Thus, this report aims to explain the modifications and the decisions made during the development process.

Chapter 2

Development

This chapter intends to explain the advanced techniques incorporated in order to improve the domino-game software.

2.1 Design Patterns

Software designing is not an easy task. Including design patterns make the code more neat. They allow to organize the structure of the code in a way that improves readability, extensibility and maintenance. At the moment of using them it is essential to think about all the possible solutions and select the one that suits the best the current problem.

On the domino game, a design pattern was introduced:

2.1.1 Singleton

This design pattern ensures that a class has only one instance and provides a global point of access to it. It can be applied when applications needs only one instance of an object.

This circumstance occurs on the board and the stock of the game. Since the objects are shared by the players, only one instance of each is needed.

The implementation used on theses classes allow the creation of threads without threatening the one-instance feature of the class.

Before including the design pattern, each time the player pressed the *enter key*, a new object *Table* and a new object *Stock* were created. Therefore, the initial amount of dominoes could continue to be drawn and shown into the graphical interface

indefinitely. After its incorporation, since only one instance of the stock and the table can be created, if the user pressed the enter key *enter key*, initial-amount of dominoes would be drawn from the stock until it was empty. Nevertheless, this behavior changed after adding invariant and throwing exceptions to the classes. (See section 2.3).

2.2 Generics

Generics allow to create reusable code. In this stage, they were used to represent the dominoes. During the lectures, one of the task was to create a generic class to represent a pair of objects. Based on this abstraction, a *GenericDomino<T>* class was created as an extension of the class *GenericPair<T>*. This allows to create dominoes of any kind, they can be dominoes of integers, characters, booleans, images and any object needed.

Since GenericDomino was too general. The class *DominoInt* was created as an extension of a *GenericDomino*. This admit the inclusion of specific constant values to represent a correct state of a domino according to the specifications. For example, the minimum value of a side of the domino was established as zero and the maximum value as six.

This class also implement *InterfaceDomino*, an interface provided as part of the graphical interface, in order to be able to use it. It was necessary to implement *getLeftValue()* and *getRightValue()* methods.

In case of changes on the requirements, if we would like to include a domino set of a different (such as dominoes of matching words, images, or a larger range of number) would not imply big changes on the current structure of the game.

Potential generic structures can be added to the game to represent the hand of the players, the stock and the board. It would improve the re usability of the code, and would allow the inclusion of different types of objects inside the game in a very simple way.

2.3 Exceptions

Exceptions are thrown when a method encounters an abnormal condition. Exceptions may be caught, but for the moment, only the throwing them was considered. Constant values indicating the correct state or behavior of the classes were added, they allowed a simple implementation of the exceptions. (See subsection 2.5.3)

2.3.1 Illegal Argument

This exception indicates that the arguments to a method call are not valid. It was included on *DominoInt*, *Table*, *Stock* and *Game* classes. Some examples are: when trying to add null dominoes to the board or to the stock, when a state number out of range is intended to be assign, when the values set to a domino is out of the valid range.

2.3.2 Illegal State

This exception indicates that a method call is not valid in the current state. It was included on *Table*, *Stock* and *Game* classes. Some examples are: when trying to remove a domino from an empty table, when trying to draw a domino from an empty stock, or when trying to call the method that executes the computer actions (*void computerPlay()*) while it is the player's turn.

2.3.3 Invariant Broken

This exception indicates that a method call breaks the invariant of the object executing the method. It was included on *Table*, *Stock*, *DominoInt* and *Game* classes. In order to accomplish this task, it was necessary to create an appropriate method *boolean invariant()* that verifies conditions defined on the requirements of the games.

The invariants for the board and the stock check the size and the invariant of each domino contained.

The invariant for the domino checks that the values of the left and the right side are inside the range designated (0 and 6).

The invariant for the game checks that the sum of the amount of dominoes distributed on the hands of the players, the stock and board is always equal to the total amount of dominoes on the game (28). It also verifies that the invariant of the stock and the table are kept.

For this exception, the code provided by J. Paul Gibson was used.

2.3.4 Bad Match

This exception indicates that a domino was added to the table but it does not match an end side of the board. It was included on *Table* class. The code used was provided by Sophie Chabridon on the graphical interface packet. Nevertheless, it was modified, in order to make it unchecked, this way it would not need to be caught.

2.4 Automated documentation

Automated documentation was added using Javadoc. Even though, Javadoc-style comments were added on the first version of the software (only *@param* and *@return* tags), these comments were improved by adding the proper *@throws* and *@link* tags, and increasing the quality of the comments in meaning and explanations. Improvements on comments were done during the whole developing process.

2.5 Other improvements

2.5.1 Code refactoring

The existing code suffered some changes without modifying the main behavior. This allowed to improve the readability of the code. Modifications were done on variable names, methods names and for loops that iterated over the lists. Besides, code lines that imported packaged never used were removed.

2.5.2 Documentation update

Documentation of the software was constantly updated during the whole development process of this version of the game. Every time that a method was added or deleted, when a design pattern was introduced and when structures related to generics were included, proper comments were created and the class diagram was brought up to date in order to make the code consistent with the design specification.

In addition, the rules of the game were included into the documentation. This document was inspired by the supplied initial rules of the game. The file was extended and it describes more precisely the behavior of the game than the one provided. This document acts as requirements of the software.

A flow chart was also included. This specifies the behavior of the game and the relation between the different states (which at the beginning where specified by numbers, and on this version constant values were used). It makes the code more understandable.

2.5.3 Good habits

Numbers used in the code were replaced by constant values (specified as static final). This fact ease the maintenance and readability of the code. Especially on the *Game* class, because a new constant value was introduced for each state of the

game.

Furthermore, new constant values were added on the classes *Game*, *Table*, *Stock* and *DominoInt*. These values represent the maximum size, minimum size, amount of pieces, and other characteristics considered constant on each class. These attributes aid the implementation of invariant methods on the same classes (which is also considered a good habit).

Finally, *toString* methods were introduced on classes *Table* and *Stock*.

2.5.4 Developing process

This time, some software engineering techniques were used:

2.5.5 Version control

Evolution of this version can be tracked on a github repository [1]. Several branches were created to develop the features added. But 2 of them are considered principal:

Master branch: stores the official release of the project.

Development branch: it is the integration branch. It is the result of the merge of the branches were the features were built.

2.5.6 Eclipse

In this occasion, Eclipse was used. It really simplified the development process. Correcting the syntax immediately, doing suggestions and even checking the orthography on the comments. This tool was specially useful during code refactoring and generating the Javadoc.

2.5.7 Task planning

The expected time and the actual time taken on each task was recorded. In order to learn the developing rhythm and performance. On this case, expected time was calculated more accurately than last semester. (See Appendix A)

2.5.8 Code reuse

The code generated during the lectures was reused, especially the one used to implement generics. In addition, the code that specifies the *InvariantBroken* exception (subsection 2.3.3) created by J.Paul Gibson was included on the game code.

Chapter 3

Conclusion

Some of the advanced techniques introduced during the course were implemented. This project allowed to know better some Java features and to experience object-oriented programming.

It was also a good opportunity to apply the working techniques learnt during the course of software engineering during the first semester, such as version control, task planning and code reuse.

This version of the project focused on improving the quality of the design and structure of the code, aiding maintenance and adding re usability. Not in finding bugs or inaccurate behaviors on the software.

The code of the game can still be improved. Among the changes that can be done there are:

- Catching exceptions.

- Implement generic classes to hold the stock, hands and board of the game.

- Using a structure that guarantees that the dominoes are not repeated inside the stock, the hand, or the board such ass a *HashSet*.

- Improve the invariant on the *Game* class, such that no domino is repeated on all the possible containers of the game.

- Inclusion of other design patterns such a MVC pattern.

- Creation of unit test.

The main difficulty during this step, was the absence of test on the code. Every time something was changed it was necessary to play the game several time, trying to get to specific situations, in order to check if something was broken.

Bibliography

[1] Patricia REINOSO. *Domino Game Repository*. `https://github.com/patriciareinoso/dominoes-game`. 2017.

Appendix A

Appendix: Tasks

Table A.1: Task planning table.

Task	Activities	Expected time	Actual time
Design Patterns	Implement singleton pattern on the Table class	2 hours	1 hour
	Implement singleton pattern on the Stock class	2 hours	0.5 hours
	Class diagram update	1 hours	0.5 hours
Generics	Implement GenericPair class	2 hours	2 hours
	Implement GenericDomino class	2 hours	2 hours
	Update DominoInt class	2 hours	2 hours
	Class diagram update	1 hours	1.5 hours
Exceptions	Add exceptions on DominoInt class	1 hour	3 hours
	Add exceptions, constant values on Hand class	1 hour	2.5 hours
	Add exceptions on Player class	1 hour	0.5 hours
	Add exceptions on Stock class	1 hour	1 hour
	Add exceptions on Table class	1 hour	1 hour
	Add exceptions on Game class	1 hour	1.5 hours
	Class diagram update	1 hour	1 hour
Automated Documentation	Add appropriate comments	4 hours	6 hours
Others	Code refactoring	1 hour	1.5 hours
	Requirements document update	1 hour	1 hour
	Flow chart creation	1 hour	2 hours
Total		21 hours	29 hours